Report from a Preliminary Investigation into LArSoft Performance

Rob Kutschke, FNAL SCD

March 11, 2016

CS-doc-5708

This report has been prepared in response to the following request from Panagiotis Spentzouris, head of SCD:

> I hear from many sources that the native Kalman Filter in LArSoft has both performance issues (in terms of fidelity, not speed) and appears to be not well put together. I would like to ask you to work with the LArSoft team (through Ruth) to get your group to review this code and provide recommendations for improvements.

In response to this request I conducted a preliminary investigation to determine how to organize a review. This report gives the results of the preliminary investigation. The bottom line is that the report identifies some issues that should be addressed by the LArSoft and Kalman Filter teams (I understand that the Kalman Filter team currently consists of Herb Greenlee and Tracy Usher). After the teams have addressed these issues we can discuss how to organize a more formal review. I do not believe that it would be useful to organize a formal review before those are addressed.

The preliminary investigation had two parts:

A. I sat in on a few meetings of a project that was already in progress: under the guidance of Marc Paterno and Jim Kowalkowski, Saba Sehrish and Gianluca Petrillo are working to refactor Track3DKalmanHit_module.cc so that the revised code can be used as model of recommended practices for writing a module in which a framework-agnostic algorithm object interacts with the LArSoft event model and the art/LArSoft services.
B. I drilled down into a few parts of the code that had piqued my interest during part A.

The report is organized into 3 parts.

1) Part 1 repeats the information transmitted by email on Feb 8, 2016; this is the recommendation for the Kalman Filter team to develop a final fitter algorithm.
2) Part 2 is new with this report. It makes recommendations that address how the code is “put together”.

3) An appendix contains 29 comments on the organization of the code and on software engineering issues. I have looked at only a small fractionof the code and I am concerned that this may be only the tip of the iceberg.

Part 1:

1) I have conducted a preliminary investigation to determine how to organize the review; in the process I identified a few known issues.
2) Working with the LArSoft, Pandora and Kalman teams, I have identified a piece of work that I think will go a long way towards resolving the known issues. See bullet 4.
3) I recommend that the work described in bullet 4 be done. After it is complete I recommend that we revisit the question of whether or not a review is needed.
4) I recommend that the Kalman filter team, the Pandora team and the LArSoft design team work together to implement the following. The Kalman filter team should support using their code as a final fitter - by this I mean that their input should be an ordered set of hits plus an initial estimate of the 3D trajectory. Their code should simply run a Kalman filter on this input and report the output. The Pandora team should learn to use the Kalman filter in this configuration and evaluate if it resolves the known physics performance issues. The transmission of the additional information between the two codes may involve the addition to or modification of LArSoft data products and LArSoft interfaces; the LArSoft design team needs to be part of this discussion in order to ensure that the solution is compliant with LArSoft standards and recommended practices. The three parties should discuss among other things:
    a. Who is responsible for adding surfaces at which there are missing hits?
    b. Should the Kalman filter do any outlier rejection? If so, this should be under the control of run-time configuration.
    c. As knowledge of the trajectory is evolves, the Kalman filter may come up with a slightly different hit ordering than that transmitted by the Pandora team. The parties should discuss the desired action of the Kalman filter in this case.

Part 2:

Findings:

1) Track3DKalmanHit_module.cc has three operating modes; the mode control is scattered throughout the code.
2) In Part 1, I made a recommendation to implement a "final fitter" mode for Track3DKalmanHit. I now know that most of this work needs to be done inside KalmanFilterAlg.cxx, not inside the Track3DKalmanHit module. As such it is not within what I understand to be the current scope of Saba's project.
3) Track3DKalmanHit has a mode in which it reads in information from PFParticles. I don't understand the intent of this mode: it discards much of the information available from the PFParticle and redoes the pattern recognition.
4) The LArSoft team has not had the resources to review code that has been contributed to LArSoft, either for physics fidelity or for good software engineering practices. This has lead to a bad situation that is described in findings 5), 6) and 7). Erica is aware of these issues at the big picture level and is aware of many in detail.
5) There are several heavily used and highly visible data product classes that are badly designed; as a result they are both difficult to use and computationally inefficient. Some details are presented in the Appendix.
6) There are many parts of Track3DKalmanHit, and the code that it uses, that are written in ways that are needlessly slow to execute, use a needlessly large amount of memory and are written in a way that invites trivial errors. I have noted 29 of these issues in Appendix I.
7) The issues noted in the Appendix were noticed during a short, quick look at small fraction the LArSoft code. If the issues described in the Appendix are representative of the code base as a whole, then LArSoft may be dying a death of 1000 cuts. If this is indeed the case, profiling the code and attacking hot spots will only get LArSoft part way to optimal performance; the only way to get all the way there is by a systematic internal review of all code.

Recommendations:

1) The LArSoft team should split Track3DKalmanHit into three algorithm classes, one for each mode. All three classes should use a common set of underlying tools that are extracted from Saba's refactoring work. This is a reasonable extension of Saba's project and should take less than one week.
2) When the final fitter is implemented, it should be organized as a 4^{th} algorithm class that, as much as possible, uses the same underlying tools as the first

three. In the appendix there are some comments on ways to improve some corners of KalmanFilterAlg.cxx; addressing those issues should be part of this work. There are two ways to get this work done. One is for the Kalman filter team to do it. The other way is to decide that someone outside of the present Kalman filter team needs to be trained for a long term support role; this would be an appropriate initial project for such a person but the learning curve will be very long.

3) All relevant stakeholders should decide if the current PFParticle mode of Track3DKalman fit needs continued support or if it should be removed; the LArSoft team leadership should develop the list of relevant stakeholders.
4) The LArSoft team should address the issues identified in Appendix I.
5) I do not believe that organizing a more formal review involving people outside of LArSoft would add value at this time. Instead the LArSoft team should implement the recommendations of this preliminary investigation. As that effort matures we can reevaluate the need for an external review.
6) In light of findings 5), 6) and 7) I recommend that the LArSoft team develop a plan to implement an internal peer review process for all LArSoft code; this should address both physics and software engineering issues. In internal review team should be empowered to request support from outside of LArSoft, both on physics issues and on software engineering issues. Think of this as an analog to the internal review process within a physics working group: it is both a quality control mechanism and an important element of training. For this to work, there needs to be buy-in from all stakeholders. Given the limited human resources, it will take a long time address the backlog of code already committed. I have specifically recommended an INTERNAL review process that may, from time to time, request external consulting; the scope of work that is needed is much larger than can be handled by an external review.
7) Once the LArSoft team has developed the plan requested in recommendation 6), the LArSoft stakeholders should decide on how to set priorities and how to staff the peer review process.

Appendix:

Executive Summary:

This appendix calls out some specific issues that I noticed in the course of the preliminary investigation. They appear in the order that they were noticed and they fall into 3 categories:

A. Insufficient big-picture thinking
B. Unfortunate physics decisions (e.g. discarding information that Pandora produces)
C. Poor use of C++: many of these issues involve poor choice of data structures, leading to inefficient and hard-to-maintain code; here inefficiency refers to both wasting CPU time and having a larger than necessary memory footprint.

List of Issues:

1) There is a lot of code in which 3-points, 3-momenta, and directions in 3-space are stored as c-style arrays of length 3:

```
double point[3];
```

The LArSoft project should decide on a preferred 3-vector class and use it everywhere.

Examples are in the classes, Seed, KTrack, KGTrack and Surface (and all classes derived from Surface); there are also examples within

```
Track3DKalmanHit_module.cc
```

Be aware that choosing ROOT's classes for this purpose has a downside that should be understood and included in the decision making process; the downside is wasted memory because of TObject-ness. This is discussed more in item 14.

If it is not possible to make this change quickly, it would be a good idea to provide a library that does simple algebra on 3-vectors implemented as c-style arrays (addition, subtraction, magnitude, perpendicular component, dot and cross products). Consult with software experts to decide which of these functions should be inlined.

When deciding on a preferred 3-vector class, the LArSoft team should take a bigger view and include:

a. a preferred 4-vector class

b. a preferred 3-rotation class
c. a preferred 4D Lorentz transformation class
d. Think about interfaces to third party products: if many third party products use the same package, then it might be wise to adopt that package as the LArSoft standard; this would remove the need for transformation of representations at package boundaries - for example at the LArSoft-Geant4 boundary.

An example of d) is Mu2e's decision to use the CLHEP 3-vector and rotation classes in its geometry system; Mu2e also chose to use the Geant4 system of units. The result is that there are no translations needed at the Mu2e/Geant4 boundary.

There is no reason to couple a decision on linear algebra classes to this decision.

2) There are many accessor member functions that return their information by copying it into an argument supplied by the caller. For example, the following from Seed.h:

```
void GetPoint(double* Pt,  double* Err) const;
```

The preferred implementation is to return this sort of information by const& or const* to an underlying 3-vector class.

Examples are at the same places as 1).

Consider the class Seed. The authors did a good job of encapsulating internal representation so - so it will be easy to change the internal representation, to an appropriate 3-vector class. They can then add new public interface and keep all of the old public interface. Therefore the change can be done gradually – eventually the old interface should be removed.

3) The class Seed has an accessor member function with the signature:

```
void GetPoint(double* Pt,  double* Err) const;
```

There is no accessor that returns only the point, not it's error.

There code in Track3DKalmanHit_module.cc and KalmanFilterAlg.cxx that needs access to the point but does not use the error.

This requires wasteful filling of the unused return argument.

This is automatically fixed by fixing 1 and 2. If these steps cannot be taken promptly, consider providing and using a new accessor:

```
void GetPointOnly(double* Pt ) const;
```

Examples are in the same places as 1)

4) There is frequent inappropriate use of pow. For example in Seed.cxx:

```
double Seed::GetLength() const  {
  return pow(pow(fSeedDirection[0],2)+
         pow(fSeedDirection[1],2)+
        pow(fSeedDirection[2],2),0.5);
}
```

This code computes the magnitude of a 3-vector; it is needlessly obfuscating and likely is needlessly slow.

Some years ago the ATLAS people did a study of a particular versioncof gcc and concluded that pow(x,2) compiles into x*x; similarly for other small integer powers that are known at compile time. I have no idea if pow(x,0.5) compiles into sqrt(x) or into exp(0.5*log(x)). So it's possible that on some compilers there is no speed issue; but it seems silly to count on all relevant compilers being so aware.

Even if there is no speed issue this style of coding error prone and hard to parse;

This problem would be solved by fixing issue 1), in which case the above example reduces to a call to a magnitude function.

As a side factoid, cetlib supplies

```
$CETLIB_INC/cetlib/pow.h
```

which implements a class template to compute all non-negative powers of its argument using the minimum multiply algorithm:

```
#include "cetlib/pow.h"

  double x{5};
  const unsigned n{6};
  double y2 = square(x);  // 5.^2 =  25.
  double y3 = cube(x);    // 5.^3 = 125.
  double y4 = fourth(x);  // 5.^3 = 625.
  double y6 = pow<n>(x);  // 5.^6;
```

In the last line, n must be a compile time constant. This package also provides another handy free-function template to compute the difference of squares:d

```
template< class T >
inline T
  cet::diff_of_squares( T x, T y )
```

```
{ return (x+y) * (x-y); }
```

Too often I see a*a-b*b written exactly like that, when it should always be written as (a+b)*(a-b). Basically, we usually get away with this, until we don't.

5) In KalmanFilterAlg.cxx there is very wasteful and unnecessary copying. This can be speeded up by choosing an alternate bookkeeping mechanism. The details are below.

The member functions buildTrack and extendTrack take a collection of hits as in input argument; these are hits that the code is free to use to build or extend the track. There is no provision for hits to be flagged as available or unavailable; the code presumes that all hits are available.

Internal to KalmanFilterAlg, there is used/free bookkeeping but this is not exposed outside of the class.

This design has a very important negative side effect. The cartoon picture of part of Track3DKalmanHit is that it starts with a bucket of hits represented by:

```
art::PtrVector<recob:Hit>
```

It then transforms these hits into a new representation of type:

```
KHitContainer
```

I understand that there are frequently many thousand hits in each container.

Then the code calls member functions of KalmanFilterAlg that operate on the KHitContainer. After each call Track3DKalmanHit needs to:

a) Interrogate the output of the member function call to discover which hits are now on the track.
b) Compress the art::PtrVector<recob:Hit> to remove the hits now on the track
c) Build a new KHitContainer object from the compressed list

Then it makes the next call to a member function of KalmanFilterAlg. This is repeated up to 6 times for each track.

Moreover, the compression algorithm in step b) needlessly re-sorts an already sorted container; this is done up to 6 times for each track. The bottom line is that there is expensive compression of an art::PtrVector followed by an expensive recreation of the KHitContainer. This is done many times within processing a single seed track.

If the used/free bookkeeping within KalmanFilterAlg were made available to the outside this bookkeeping could be coded in a more readable fashion and the code would execute more quickly. Marking hits as used is much faster than all of the manipulations done currently.

6) In the Pandora mode of Track3DKalman, the code reads in tracks from the event and builds internal representations of the information from the event. It then processes one track at a time and builds an internal representation of the output for each track. When all is done, loops over the output tracks to fill histograms and then it translates its internal output representation into data products. At no point does it ever consider two input tracks at the same time.

This code could be rewritten to:

- Read the next input from Pandora
- Process that input
- Make histograms for tracks found in this step
- Add the output from this step to the data product
- Let all transient information associated with this step go out of scope.

This will reduce the transient memory footprint.

7) recob::Track has several serious design flaws. Here is an easy one:

A track covariance matrix is symmetric. The class used to represent it in recob::Track is general 5x5 matrix class, which has 25 elements, not the necessary 15.

Therefore 40% of the memory footprint and 40% of the disk space are deadweight loss.

8) The biggest flaw in recob::Track is it that exposes a large number of implementation details that take significant expertise to use correctly. Instead the class should present an intuitive face that hides the implementation details better.

Some examples of inappropriately exposed details are given below. Here are some of the data members of recob::Track:

```
std::vector<TVector3> fXYZ;  // position of points along the track
std::vector<TVector3> fDir;  // direction at each point along the track
std::vector<TMatrixD > fCov; // covariance matrix of positions
                             // possibly only end points are stored
std::vector<double> FitMomentum; // momentum at start and end of track
                                 // determined from a fit. Intermediate
                                 // points can be added if desired
```

The various collections can be of different lengths and this is exposed to the end user, without protection. Basically you just gotta know.

Suppose that you have a track with N hits; it is legal for there to be N values of fXYZ and fDir but only two values for fCov and fFitMomentum. You just gotta know to check the size of fCov before you use it.

There are well known ways to design an interface that protects end users from this sort of ambiguity and many other internal implementation details – some of which are discussed below.

A full discussion of this is beyond the scope of this report.

9) Another issue with recob::Track. The comments say that fCov is the covariance matrix of the position, which I interpret to be a 3x3 covariance matrix. When I look at the code for KGTrack.h is see that it actually puts a 5x5 covariance matrix into the recob::Track.

Is this just a misleading comment or are there other producers of recob::Track objects that produce differently dimensioned covariance matrices? Or even worse, produce 5x5 covariance matrices with different basis variables?

This means that in order to understand the meaning of the covariance matrix you need to know which code that produced it. This is a dangerous situation.

For tracks produced by Track3DKalmanHit, I think that the 5 variables in the covariance matrix are 2 slopes, 2 intercepts and the magnitude of the momentum.

This is not documented anywhere in Track.h, either directly or indirectly. You just gotta know.

10) In the parts of the code that I have seen so far, there is no way to persist "hit-on-track" objects, that have things like the final fully corrected value of the measurement, its error, the final fully corrected value of the predicted measurement, and its error. From this one can compute the residual and its error.

This information is needed for calibration and alignment. It is a really bad idea to attempt to rebuild residuals from other information in the event - almost always this produces an inferior result.

For reasons of disk space sanity it is important to design a system in which you can choose not to persist this information when it is not needed.

When having this discussion, remember that there are good use cases for computing residuals with the local hit excluded from the fit or included in the fit. Design a system in which it is possible to do one or the other or both; it should be a run time configuration decision which is chosen.

I would not use an Assns to persist this information. My own preference would be to store hit-on-track objects in a separate data product that is accessed by lock-step indexing with the track data product.

11) Track3DKalmanHit builds KGTrack objects. It calls KGTrack::fillTrack to copy the KGTrack into the recob::Track that will be the data product.

This code does does 4 copies of each element when simple additions to the public interface of KGTrack and recob::Track would allow it to be done with a single copy of each element:

Here is a cartoon:

```
void fillTrack ( recob::Track& track, /* more args */ ){

    std::vector<TVector3> xyz;
    xyz.reserve(fTrackMap.size());

    loop over hits on the track {
      double pos[3];

      // First copy
      trh.getPosition(pos);

      // Second and third copy
      xyz.push_back(TVector3(pos[0], pos[1], pos[2]));
}

    // Fourth copy in c'tor
    track = recob::Track(xyz, /* more arguments */);
  }
```

There is no fifth copy into the return value; that is optimized into a move.

To improve this KHitTrack should hold its position by an appropriate 3-vector type and have an accessor that returns the positin as a const& or const*. The other change is that recob::Track needs a move aware c'tor. With these two changes the loop can read:

```
 loop over hits : {
    Favorite_3Vector_type const& pos =  trh.getPosition();
    xyz.emplace_back( pos[0], pos[1], pos[2] );
   }
```

```
track = recob::Track( std::move(xyz), /* more args */);
```

This version does a single copy, which takes place at the emplace_back into xyz. The last line will be optimized into a move. The net result is a single copy plus one move, saving 3 copy operations.

The same issue exists for each of the five collection-type data members of recob::Track. In the current code, there is transiently one complete extra copy of all of the track data in memory. This is not needed.

12) More about KGTrack::fillTrack. Copying the covariance matrix to the recob::Track has the same issues as 11 but it also has one more. There is a mode in which the covariance matrix is saved for only the first and last points on the track.

The algorithm is:

a) For the first two hits on the track, copy the covariance matrix to the output std::vector<TMatrixD>.
b) For all later hits, overwrite the second element of the output std::vector<TMatrixD>.

Moreover, the operation of doing each wasted copy copy actually does 2 wasted copies.

The code should be written to just push_back the covariance matrices for the first and last points.

I also note that there is no call to reserve for the local variable.

```
std::vector<TMatrixD> cov;
```

This is a minor issue if the code is in the mode that it only stores the covariance matrix for the first and last hits. On the other hand, this is a serious problem if the code is storing the covariance matrix for each hit out of O(1000) hits.

13) More about KGTrack::fillTrack. There is code fragment:

```
if(parthit.get() != 0) {
         const recob::Hit& arthit = *parthit;
         geo::View_t view = arthit.View();
         double pitch = geom->WirePitch(view);
         double charge = arthit.PeakAmplitude();
         double dudw = trh.getVector()[2];
         double dvdw = trh.getVector()[3];
         double dist = pitch *
              std::sqrt(1. + dudw * dudw + dvdw * dvdw);
         double qdist = charge / dist;
         dqdx.at(view).back() = qdist;
}
```

This code is exposed at too high a level - it belongs as a member functionsomewhere but I am not sure in which class.

14) In recob::Track, points and directions are stored as type TVector3. The data payload of a TVector3 is three doubles. But it inherits from TObject - so it has an additional payload of 2 ints. In other words, 25% of the memory used by these objects is a dead weight loss. I expect that these two ints will compress well so, if the TVector3 objects are written to disk in split format, they are not likely an important part of the size on disk; but it does add to the CPU time associated with IO, and buffer sizes associated with IO.

The TMatrixD objects stored in recob::Track have the same problem but their bigger problem is storing the full 5x5 rep of a symmetric matrix. Again the extra words are likely to compress well and not cost size on disk.

Depending on how the IO subsystem is configured, it is possible to drop the TObject part of the data payload when writing an output file. This is probably not worth doing since the extra int's compress well. If we wanted to do it, this is done by making the following calls very early in the program:

```
TClass::GetClass("TVector3")->IgnoreTObjectStreamer(true);
TClass::GetClass("TMatrixTBase<double>")>IgnoreTObjectStreamer(true);
```

It's not clear to me if any user code is called early enough to put these calls into user code. It would also affect other instances of these classes in other contexts.

It's not likely that this is a good idea but I wanted it on the record.

15) This is an aside on 14). It's possible that the TObject parts of TVector3 are necessary for EventDisplays that lets users pick objects. If that is true, then this is one of the classic arguments for having separate data types for reco use and for event display use - in a reco world, that is often memory constrained, there is no sense to carry around the extra payload that is needed only for graphics.

The big picture is that event display 3-vector objects should be constructable from reco 3-vector objects; the reco objects should be lean and the event display objects may be as complex as they need to be.

16) The data product classes have too much functionality. They should be stripped down to their payload plus functions that operate on that payload. Additional functionality can be added by use of the facade pattern.

For example KGTrack has functionality that requires access to the geometry and to propagators. This does not belong in a data product class. It also complicates the use of the data product in environments other than LArSoft.

17) In KTrack.cxx

```
namespace {
  const double mumass = 0.105658367;  // Muon
  const double pimass = 0.13957;      // Charged pion
  const double kmass = 0.493677;      // Charged kaon
  const double pmass = 0.938272;      // Proton
}
```

This should be arranged so that all program units that need these masses get them from a single authoritative source. One solution is a particle data table made available as a service provider; one can also imagine build-time solutions and other run-time solutions. When thinking about this, consult with the software experts to anticipate a later transition to a multi-threading environment.

19) The KalmanFilterAlg.cxx class has a lot graphics code included. On the one hand I really like this - if you can animate pattern recognition it is a great way to understand corner cases; it is also a great way to describe to others how the algorithm works. However I would refactor the code to hide all of graphics behind function calls; this will make the algorithm proper much easier to understand, improve and maintain; it also makes it possible to "dummy out" the graphics at .so load time, thereby reducing the memory footprint of production code that does not need graphics.

20) In KGTrack the fsurf data member is of type:

```
std::shared_ptr<const Surface>
```

None of the implementation uses any of the features of shared_ptr. It would be completely OK to hold it as type:

```
Surface const* ;
```

It's very likely a good idea to get rid of the set functions, to enforce single phase construction; if that is done, it can be held as:

```
Surface const&;
```

I don't see the use case for holding it as shared_ptr - this would only make sense if there was a danger of the pointee going out of scope during the lifetime of the KGTrack object. If this happens there is a very bad design elsewhere.

21) Near the beginning of Track3DKalmanHit::produce, there is a call:

```
fSpacePointAlg.clearHitMap();
```

This call is only done on entry to produce. This means that when the module exits, the hitmap is still occupying space. This is just a waste of space.

The clearMap function should be called as soon as possible after fSpacePointAlg is no longer used for processing the current event. I think that this is is just before you move the output data products into the event.

The call needs to also remain at the start of the event. The reason is that if Track3DKalman hit throws an exception and if art is configured to continue, on that sort of exception, then it is possible that fSpacePointAlg will contain stale hitMap entries when the next event starts. Therefore it must be cleared at the start of every event; the waste of doing it both at the start and end of each event is not worth worrying about.

22) LArSoft makes heavy use of art::Assns. Based on the parts of the code I have seen, end users are expected to be able to muck around in the guts. LArSoft should provide navigator classes that provide a coherent interface to an a track and it's ensemble of associated data products, without the need for an end user to understand the details of how everything is glued together.

This comment is in the same spirit as the comment 8 - that recob::Track also lacks a coherent interface.

23) In Track3DKalmanHit, at the end of the "extend and smooth" iteration, there is a line:

```
trg1 = trg2;
```

where both variables are of type KGTrack and I understand that KGTracks may often contain many hundreds of hits. After this line, trg2 immediately goes out of scope. So this should be written as:

```
trg1 = std::move(trg2);
```

which tells the compiler to do a move, not a copy. For a track with many hundreds of hits, this is a huge savings.

However the above fix won't actually work, and the compiler will not actually do a move – it will silently do a copy instead. The issue is that KGTrack contains a user written d'tor. The d'tor is a do nothing d'tor. Simply removing it will allow the compiler to properly optimize this operation.

The copy wastes CPU and transiently doubles the memory footprint.

24) Following up on 23. The full code base should be scrubbed for unnecessary user supplied destructors, copy constructors and copy assignment operators; the presence of these will stop the compiler from automatically generating move optimizations.

25) In TrackKalman3DHit. Why there is a smooth after every extend? Is this used to verify sort order of the 2D hits added to date? Something else? I had expected to see one smooth after the full track was fitted one direction.

26) About KGTrack. The authors should consult with software engineers about the use of std::multimap as the primary container. It's likely that another solution would be more efficient computationally.

27) About TrackKalman3DHit. If you run TrackKalman3DHit with fdoDeDx true, then the output data product will, in general, contain two recob::Track objects for every real track. The two objects correspond to the two different hypotheses about the propagation direction of the track. I understand the use case in which both are needed. What I don't see is how one can tell from the data product which of two objects correspond to the same physical track (sure you could interrogate their hits but there should be an obvious "flag" visible at a high level).

Moreover, if, for one real track, one of the fits fails, then you can no longer count on the pattern that the paired tracks are (0,1), (2,3) and so on.

28) I noticed code that contained both reconstruction algorithms and code that used MC truth information to check the results of algorithms. I understand that in the earlier stages of algorithm development it may be necessary to have MC checking mixed in with reco code - and perhaps this is the case with the code that I saw.

I recommend that, as soon as possible, these functions be split so that reco code, proper, is always MC blind. This makes it a little more awkward to use the MC truth to validate the reco code but it's worth the effort: it keeps the reco code leaner for use on real data, which can be important in online environments and on the grid, both of which are memory limited.

29) Tracy Usher suggested that the LArSoft team should revisit the inheritance hierarchy to identify interfaces for which it is unlikely that there will ever be more than one concrete class. I agree that this is a worthwhile exercise and it's likely that some can be removed.